# Customer Charge End-to-End Flow Documentation

## Overview

This document provides a comprehensive breakdown of the customer charge processing flow from the initial controller action through all three AWS Lambda phases, including every internal method call and database operation.

## Phase 1: Charge Enqueueing (CustomerChargeController → AltaworxRevAWSEnqueueCustomerCharges)

### 1.1 Initial Controller Trigger

#### Entry Points:

* **Single Instance**: CustomerChargeController.Create(Guid sessionId, long id)
* **Multiple Instances**: CustomerChargeController.CreateConfirmSession(Guid sessionId, string selectedInstances, ...)

#### Controller Flow - Single Instance:

CustomerChargeController.Create(sessionId, id)  
├── Validate permissions (ModuleEnum.CustomerCharge)  
├── Get AWS credentials from custom fields:  
│ ├── AwsAccessKeyFromCustomObjects(customObjectDbList)  
│ ├── AwsSecretAccessKeyFromCustomObjects(customObjectDbList)  
│ └── CreateCustomerChargeQueueFromCustomObjects(customObjectDbList)  
├── Validate AWS setup  
└── EnqueueCreateCustomerChargesSqs(id, awsAccessKey, awsSecretAccessKey, createCustomerChargeQueueName, altaWrxDb, tenantId, 1, 1, id.ToString())

#### Controller Flow - Multiple Instances:

CustomerChargeController.CreateConfirmSession(sessionId, selectedInstances, ...)  
├── Validate permissions and setup  
├── Parse selectedInstances into instanceIds array  
├── For each instanceId:  
│ ├── Determine isLastInstanceId flag (1 if last, 0 otherwise)  
│ └── EnqueueCreateCustomerChargesWithSesstionSqs(instanceId, awsAccessKey, awsSecretAccessKey, createCustomerChargeQueueName, altaWrxDb, tenantId, 1, isLastInstanceId, string.Join(",", instanceIds))

### 1.2 SQS Message Creation

#### EnqueueCreateCustomerChargesWithSesstionSqs Method:

EnqueueCreateCustomerChargesWithSesstionSqs(instanceId, awsAccessKey, awsSecretAccessKey, createCustomerChargeQueueName, altaWrxDb, tenantId, isMultipleInstanceId, isLastInstanceId, instanceIds)  
├── Create AWS credentials: new BasicAWSCredentials(awsAccessKey, awsSecretAccessKey)  
├── Get integration authentication:  
│ ├── new IntegrationAuthenticationRepository(altaWrxDb)  
│ └── GetAuthByIntegrationId(IntegrationEnum.RevIO.AsInt(), tenantId)  
├── Create SQS client: new AmazonSQSClient(awsCredentials, RegionEndpoint.USEast1)  
├── Find queue: client.ListQueues(createCustomerChargeQueueName)  
├── Build SQS message:  
│ ├── MessageBody: "Instance to work is {instanceId}"  
│ ├── DelaySeconds: 90 if isLastInstanceId == 1, else 0  
│ └── MessageAttributes:  
│ ├── InstanceId: instanceId.ToString()  
│ ├── IsMultipleInstanceId: isMultipleInstanceId.ToString()  
│ ├── IsLastInstanceId: isLastInstanceId.ToString()  
│ ├── InstanceIds: instanceIds  
│ └── CurrentIntegrationAuthenticationId: integrationAuthentication.id.ToString()  
└── Send message: client.SendMessageAsync(request)

### 1.3 Lambda Function Handler (AltaworxRevAWSEnqueueCustomerCharges)

#### Function Entry Point:

Function.FunctionHandler(SQSEvent sqsEvent, ILambdaContext context)  
├── BaseFunctionHandler(context) → KeySysLambdaContext  
├── Get environment variable: DeviceCustomerChargeQueueUrl  
└── ProcessEvent(keysysContext, sqsEvent)

#### Event Processing:

ProcessEvent(context, sqsEvent)  
├── Validate single record: sqsEvent.Records.Count == 1  
└── ProcessEventRecord(context, sqsEvent.Records[0])

#### Record Processing:

ProcessEventRecord(context, message)  
├── Extract InstanceId from message.MessageAttributes["InstanceId"].StringValue  
├── Create SqsValues object: new SqsValues(context, message)  
│ ├── IsMultipleInstanceId: from MessageAttributes or default 0  
│ ├── IsLastInstanceId: from MessageAttributes or default 0  
│ ├── InstanceIds: from MessageAttributes or null  
│ └── CurrentIntegrationAuthenticationId: from MessageAttributes or ""  
└── ProcessInstance(context, instanceId, sqsValues)

### 1.4 Instance Processing

#### ProcessInstance Method:

ProcessInstance(context, instanceId, sqsValues)  
├── GetInstance(context, instanceId) → OptimizationInstance  
│ ├── Execute stored procedure: GET\_OPTIMIZATION\_INSTANCE  
│ ├── Parameters: @instanceId  
│ └── Return: OptimizationInstance with all properties  
├── LoadOptimizationSettingsByTenantId(instance.TenantId)  
├── GetCommGroups(context, instanceId) → List<OptimizationCommGroup>  
│ ├── SQL: "SELECT Id, InstanceId FROM OptimizationCommGroup WHERE InstanceId = @instanceId"  
│ └── Return: List of communication groups  
├── For each communication group:  
│ ├── GetWinningQueueId(context, commGroup.Id, instance.PortalTypeId)  
│ └── EnqueueCustomerCharges(context, winningQueueId, instance.PortalTypeId, sqsValues, instance.IntegrationAuthenticationId)

### 1.5 Queue Selection Logic

#### GetWinningQueueId Method:

GetWinningQueueId(context, commGroupId, portalTypeId)  
├── Determine SQL command based on portal type:  
│ ├── PortalTypeMobility (2): GetMobilityDeviceWinningQueueSql()  
│ ├── CrossProvider: GetCrossProviderDeviceWinningQueueSql()  
│ └── Default (M2M): GetDeviceWinningQueueSql()  
├── Execute SQL query:  
│ ├── Parameters: @commGroupId  
│ ├── Returns: TOP 1 Queue Id with lowest TotalCost  
│ └── Conditions: RunEndTime IS NOT NULL, TotalCost IS NOT NULL

#### SQL Queries:

**M2M Devices:**

SELECT TOP 1 Id FROM OptimizationQueue oq  
WHERE EXISTS (  
 SELECT 1 FROM OptimizationDeviceResult odr  
 WHERE oq.Id = odr.QueueId  
)  
AND CommPlanGroupId = @commGroupId  
AND TotalCost IS NOT NULL  
AND RunEndTime IS NOT NULL  
ORDER BY TotalCost

**Mobility Devices:**

SELECT TOP 1 Id FROM OptimizationQueue oq  
WHERE EXISTS (  
 SELECT 1 FROM OptimizationMobilityDeviceResult odr  
 WHERE oq.Id = odr.QueueId  
)  
AND CommPlanGroupId = @commGroupId  
AND TotalCost IS NOT NULL  
AND RunEndTime IS NOT NULL  
ORDER BY TotalCost

**Cross Provider:**

SELECT TOP 1 Id FROM OptimizationQueue oq  
WHERE EXISTS (  
 SELECT 1 FROM OptimizationMobilityDeviceResult odr  
 WHERE oq.Id = odr.QueueId  
 UNION   
 SELECT 1 FROM OptimizationDeviceResult odr  
 WHERE oq.Id = odr.QueueId  
)  
AND CommPlanGroupId = @commGroupId  
AND TotalCost IS NOT NULL  
AND RunEndTime IS NOT NULL  
ORDER BY TotalCost

### 1.6 Customer Charge Enqueueing

#### EnqueueCustomerCharges Method:

EnqueueCustomerCharges(context, queueId, portalTypeId, sqsValues, integrationAuthenticationId)  
├── Check if CrossProvider:  
│ ├── If CrossProvider:  
│ │ ├── EnqueueCustomerChargesDb(context, queueId, PortalTypeM2M)  
│ │ ├── EnqueueCustomerChargesDb(context, queueId, PortalTypeMobility)  
│ │ └── EnqueueCustomerChargesSqs(context, queueId, sqsValues, portalTypeId, sqsValues.CurrentIntegrationAuthenticationId)  
│ └── Else:  
│ ├── EnqueueCustomerChargesDb(context, queueId, portalTypeId)  
│ └── EnqueueCustomerChargesSqs(context, queueId, sqsValues, portalTypeId, integrationAuthenticationId)

#### Database Enqueueing:

EnqueueCustomerChargesDb(context, queueId, portalTypeId)  
├── Open SQL connection  
├── Execute: "SET ARITHABORT ON"  
├── Determine stored procedure:  
│ ├── Mobility: "usp\_Optimization\_Mobility\_EnqueueCustomerCharges"  
│ └── M2M: "usp\_Optimization\_EnqueueCustomerCharges"  
├── Execute stored procedure:  
│ ├── Parameters: @QueueId  
│ └── CommandTimeout: 240 seconds

#### SQS Enqueueing:

EnqueueCustomerChargesSqs(context, queueId, sqsValues, portalTypeId, integrationAuthenticationId)  
├── Create AWS SQS client  
├── Build message:  
│ ├── MessageBody: "Queue to work is {queueId}"  
│ ├── DelaySeconds: 300 if isLastQueue, else 0 (5-minute delay)  
│ └── MessageAttributes:  
│ ├── QueueId: queueId.ToString()  
│ ├── IsMultipleInstanceId: sqsValues.IsMultipleInstanceId.ToString()  
│ ├── IsLastInstanceId: sqsValues.IsLastInstanceId.ToString()  
│ ├── InstanceIds: sqsValues.InstanceIds.ToString()  
│ ├── PortalTypeId: portalTypeId.ToString()  
│ └── CurrentIntegrationAuthenticationId: integrationAuthenticationId.ToString()  
├── Send to: DeviceCustomerChargeQueueUrl  
└── Execute: client.SendMessageAsync(request)

## Phase 2: Charge Creation (AltaworxRevAWSCreateCustomerChange)

### 2.1 Lambda Function Handler

#### Function Entry Point:

Function.FunctionHandler(SQSEvent sqsEvent, ILambdaContext context)  
├── BaseFunctionHandler(context) → KeySysLambdaContext  
├── Initialize repositories and services:  
│ ├── EnvironmentRepository()  
│ ├── OptimizationInstanceRepository(logger, connectionString)  
│ ├── OptimizationQueueRepository(logger, connectionString)  
│ ├── Base64Service()  
│ ├── SettingsRepository(logger, connectionString, base64Service)  
│ ├── DeviceCustomerChargeQueueRepository(...)  
│ ├── RevioAuthenticationRepository(...)  
│ ├── RevioApiClient(...)  
│ ├── CustomerChargeListFileService()  
│ ├── CustomerChargeListEmailService(...)  
│ ├── DeviceChargeRepository(...)  
│ └── DeviceCustomerChargeService(...)  
├── Create event handler: CustomerChangeEventHandler(...)  
├── Create SqsValues: new SqsValues(logger, sqsEvent.Records[0])  
└── changeHandler.HandleEventAsync(sqsEvent, sqsValues)

### 2.2 Event Handler Processing

#### CustomerChangeEventHandler.HandleEventAsync:

HandleEventAsync(sqsEvent, sqsValues)  
├── ProcessEventAsync(sqsEvent, sqsValues)  
│ ├── Validate single record: sqsEvent.Records.Count == 1  
│ └── ProcessEventRecordAsync(sqsEvent.Records[0], sqsValues)

#### ProcessEventRecordAsync:

ProcessEventRecordAsync(message, sqsValues)  
├── Check message attributes:  
│ ├── If contains "QueueId":  
│ │ ├── Parse queueId: long.Parse(message.MessageAttributes["QueueId"].StringValue)  
│ │ ├── Get queue: \_optimizationQueueRepository.GetQueue(queueId)  
│ │ ├── Get instance: \_optimizationInstanceRepository.GetInstance(queue.InstanceId)  
│ │ └── \_deviceCustomerChargeService.ProcessQueueAsync(queueId, instance, sqsValues)  
│ └── If contains "FileId":  
│ ├── Parse fileId: int.Parse(message.MessageAttributes["FileId"].StringValue)  
│ └── \_deviceCustomerChargeService.ProcessQueueAsync(fileId, sqsValues)

### 2.3 Enhanced SqsValues Construction

#### SqsValues Constructor (Enhanced Version):

SqsValues(logger, message)  
├── IsMultipleInstanceId: Convert.ToBoolean(Int32.Parse(message.MessageAttributes["IsMultipleInstanceId"].StringValue)) or false  
├── IsLastInstanceId: Convert.ToBoolean(Int32.Parse(message.MessageAttributes["IsLastInstanceId"].StringValue)) or false  
├── InstanceIds: message.MessageAttributes["InstanceIds"].StringValue or ""  
├── PortalTypeId: Int32.Parse(message.MessageAttributes["PortalTypeId"].StringValue) or 0  
├── CurrentIntegrationAuthenticationId: Int32.Parse(message.MessageAttributes["CurrentIntegrationAuthenticationId"].StringValue) or 0  
├── IsSendSummaryEmailForMultipleInstanceStep: Convert.ToBoolean(Int32.Parse(message.MessageAttributes["IsSendSummaryEmailForMultipleInstaceStep"].StringValue)) or false  
├── RetryNumber: Int32.Parse(message.MessageAttributes["RetryNumber"].StringValue) or 0  
├── PageNumber: Int32.Parse(message.MessageAttributes["PageNumber"].StringValue) or 1  
└── RetryCount: Int32.Parse(message.MessageAttributes["RetryCount"].StringValue) or 0

### 2.4 Device Processing Service

#### DeviceCustomerChargeService.ProcessQueueAsync (Queue-based):

ProcessQueueAsync(queueId, instance, sqsValues)  
├── Calculate pagination offset: (sqsValues.PageNumber - 1) \* PAGE\_SIZE (50)  
├── Determine customer type:  
│ └── isNonRevCustomer = instance.AMOPCustomerId != null && instance.RevCustomerId == null && instance.IntegrationAuthenticationId == null  
├── Get environment variables:  
│ ├── connectionString: GetEnvironmentVariable("ConnectionString")  
│ ├── proxyUrl: GetEnvironmentVariable("ProxyUrl")  
│ └── bucketName: GetEnvironmentVariable("CustomerChargesS3BucketName")  
├── Get service providers: ServiceProviderCommon.GetServiceProviders(connectionString)  
├── Get device list: \_customerChargeQueueRepository.GetDeviceList(queueId, PAGE\_SIZE, offset, isNonRevCustomer)  
├── Filter unprocessed: deviceList.Where(x => x.IsProcessed == false)  
├── Branch processing:  
│ ├── If isNonRevCustomer: ProcessCustomerChargeForNonRev(...)  
│ └── Else: ProcessCustomerChargeForRev(...)

### 2.5 Non-Rev Customer Processing

#### ProcessCustomerChargeForNonRev:

ProcessCustomerChargeForNonRev(queueId, instance, sqsValues, deviceLists, serviceProviders, proxyUrl, bucketName, offset)  
├── Get settings: \_settingsRepository.GetOptimizationSettings()  
├── For each device in deviceLists:  
│ └── \_customerChargeQueueRepository.MarkRecordProcessed(device.Id, "0", device.DeviceCharge, device.BaseRate, device.DeviceCharge + device.BaseRate, false, "", "0", device.SmsChargeAmount)  
├── Calculate total pages: CalculateTotalPageInQueue(queueId, PAGE\_SIZE, true)  
├── Enqueue additional pages: MultipleEnqueueCustomerChargesAsync(queueId, sqsValues, totalPage, true)  
└── If last page: EnqueueCheckCustomerChargesIsProcessedAsync(queueId, sqsValues.PortalTypeId, sqsValues.InstanceIds, sqsValues.IsMultipleInstanceId, sqsValues.IsLastInstanceId)

### 2.6 Rev Customer Processing

#### ProcessCustomerChargeForRev:

ProcessCustomerChargeForRev(queueId, instance, sqsValues, deviceList, serviceProviders, proxyUrl, bucketName, offset)  
├── Validate integration authentication: instance.IntegrationAuthenticationId.HasValue  
├── Get Rev.io authentication: \_revIoAuthenticationRepository.GetRevioApiAuthentication(instance.IntegrationAuthenticationId.Value)  
├── Get settings:  
│ ├── optimizationSettings: \_settingsRepository.GetOptimizationSettings()  
│ ├── billingTimeZone: optimizationSettings?.BillingTimeZone  
│ └── useNewLogicCustomerCharge: optimizationSettings?.UsingNewProcessInCustomerCharge  
├── Process devices: ProcessDeviceList(deviceList, queueId, sqsValues, instance, revIoAuth, billingTimeZone, serviceProviders, useNewLogicCustomerCharge)  
├── Calculate total pages: CalculateTotalPageInQueue(queueId, PAGE\_SIZE, false)  
├── Enqueue additional pages: MultipleEnqueueCustomerChargesAsync(queueId, sqsValues, totalPage, false)  
└── If last page: EnqueueCheckCustomerChargesIsProcessedAsync(queueId, sqsValues.PortalTypeId, sqsValues.InstanceIds, sqsValues.IsMultipleInstanceId, sqsValues.IsLastInstanceId, sqsValues.CurrentIntegrationAuthenticationId)

### 2.7 Device List Processing

#### ProcessDeviceList:

ProcessDeviceList(deviceList, queueId, sqsValues, instance, revIoAuth, billingTimeZone, serviceProviders, useNewLogicCustomerCharge)  
├── For each device in deviceList:  
│ ├── Check new logic flag:  
│ │ ├── If useNewLogicCustomerCharge == true:  
│ │ │ ├── If device.DeviceCharge > 0:  
│ │ │ │ ├── If device.CalculatedRateCharge > 0 && (device.RevProductTypeId != null || device.RevProductId != null):  
│ │ │ │ │ └── ProcessDevice(device, instance, revIoAuth, serviceProviders, billingTimeZone, useNewLogicCustomerCharge, isRateCharge: true, isOverageCharge: false)  
│ │ │ │ └── If device.CalculatedOverageCharge > 0 && (device.OverageRevProductTypeId != null || device.OverageRevProductId != null):  
│ │ │ │ └── ProcessDevice(device, instance, revIoAuth, serviceProviders, billingTimeZone, useNewLogicCustomerCharge, isRateCharge: false, isOverageCharge: true)  
│ │ │ └── If device.SmsChargeAmount > 0 && (device.SmsRevProductTypeId != null || device.SmsRevProductId != null):  
│ │ │ └── ProcessDevice(device, instance, revIoAuth, serviceProviders, billingTimeZone, useNewLogicCustomerCharge, isRateCharge: false, isOverageCharge: false, isSMSCharge: true)  
│ │ └── Else (old logic):  
│ │ └── ProcessDevice(device, instance, revIoAuth, serviceProviders, billingTimeZone, useNewLogicCustomerCharge)  
│ └── Handle retry logic if retryFlag is true  
├── Handle error devices and retry logic:  
│ ├── If errors and retryCount <= MAX\_RETRY\_COUNT:  
│ │ └── EnqueueCustomerChargesAsync(..., retryCount + 1)  
│ └── If retryCount > MAX\_RETRY\_COUNT:  
│ └── Mark all devices as failed with final error message

### 2.8 Individual Device Processing

#### ProcessDevice Method:

ProcessDevice(device, instance, revIoAuth, serviceProviders, billingTimeZone, useNewLogicCustomerCharge, isRateCharge, isOverageCharge, isSMSCharge)  
├── Initialize variables: chargeId, smsChargeId, hasErrors, errorMessage, integrationId, statusCode  
├── Get integration ID: serviceProviders.FirstOrDefault(x => x.Id == device.ServiceProviderId).IntegrationId  
├── Validate rate plan: Check if device.RatePlanCode is not null/empty  
├── If SendToRev environment variable is true:  
│ ├── Process Usage Charges (if !isSMSCharge && device.DeviceCharge > 0):  
│ │ └── AddCustomerUsageChargeAsync(device, instance, billingTimeZone, integrationId, useNewLogicCustomerCharge, isRateCharge, isOverageCharge)  
│ ├── Process SMS Charges (if isSMSCharge && device.SmsChargeAmount > 0):  
│ │ └── AddCustomerSmsChargeAsync(device, integrationId, instance, billingTimeZone, useNewLogicCustomerCharge)  
├── Handle response:  
│ ├── If no errors and statusCode != 429 (TooManyRequests):  
│ │ └── MarkRecordProcessed(device.Id, chargeId, device.DeviceCharge, device.BaseRate, totalChargeAmount, false, "", smsChargeId, device.SmsChargeAmount)  
│ └── Else:  
│ └── Return true (retry flag)

### 2.9 Charge Creation via Rev.io API

#### AddCustomerUsageChargeAsync:

AddCustomerUsageChargeAsync(device, instance, billingTimeZone, integrationId, useNewLogicCustomerCharge, isRateCharge, isOverageCharge)  
├── LookupRevServiceAsync(device) → (revService, statusCode)  
│ ├── Get service number: device.RevServiceNumber or device.MSISDN  
│ ├── Call Rev.io API: \_revApiClient.GetServicesAsync<RevServiceList>(serviceNumber, \_logger)  
│ ├── Validate response and return active service  
│ └── Handle rate limiting (429 status code)  
├── If revService found:  
│ └── AddRevCustomerUsageChargeAsync(device, revService, instance, billingTimeZone, integrationId, useNewLogicCustomerCharge, isRateCharge, isOverageCharge)  
└── Else: Return error response

#### AddRevCustomerUsageChargeAsync:

AddRevCustomerUsageChargeAsync(device, revService, instance, billingTimeZone, integrationId, useNewLogicCustomerCharge, isRateCharge, isOverageCharge)  
├── Check logic type:  
│ ├── If useNewLogicCustomerCharge:  
│ │ ├── If isRateCharge:  
│ │ │ ├── Create request: new CreateDeviceChargeRequest(device, revService, instance.BillingPeriodStartDate, instance.BillingPeriodEndDate, billingTimeZone, integrationId, false, useNewLogicCustomerCharge, false, true)  
│ │ │ └── \_deviceChargeRepository.AddChargeAsync(requestRateCharge)  
│ │ └── If isOverageCharge:  
│ │ ├── Create request: new CreateDeviceChargeRequest(device, revService, instance.BillingPeriodStartDate, instance.BillingPeriodEndDate, billingTimeZone, integrationId, false, useNewLogicCustomerCharge, true, false)  
│ │ └── \_deviceChargeRepository.AddChargeAsync(requestOverCharge)  
│ └── Else (old logic):  
│ ├── Create request: new CreateDeviceChargeRequest(device, revService, instance.BillingPeriodStartDate, instance.BillingPeriodEndDate, billingTimeZone, integrationId)  
│ └── \_deviceChargeRepository.AddChargeAsync(request)

#### DeviceChargeRepository.AddChargeAsync:

AddChargeAsync(request)  
├── Serialize request: JsonConvert.SerializeObject(request)  
├── Call Rev.io API: revioApiClient.AddChargeAsync(requestString, retryPolicy, logger)  
├── Validate response:  
│ ├── If response == null || response.Id <= 0:  
│ │ └── Return error: new CustomerChargeResponse { HasErrors = true, ErrorMessage = errorMessage }  
│ └── Else:  
│ └── Return success: new CustomerChargeResponse { ChargeId = response.Id, HasErrors = false, ErrorMessage = "" }

### 2.10 Pagination and Re-enqueueing

#### MultipleEnqueueCustomerChargesAsync:

MultipleEnqueueCustomerChargesAsync(queueId, sqsValues, totalPage, isNonRev)  
├── If totalPage > 1 && sqsValues.PageNumber == 1:  
│ └── For pageNumber = 2 to totalPage:  
│ ├── If isNonRev:  
│ │ └── EnqueueCustomerChargesAsync(queueId, sqsValues.PortalTypeId, sqsValues.InstanceIds, sqsValues.IsMultipleInstanceId, sqsValues.IsLastInstanceId, pageNumber)  
│ └── Else:  
│ └── EnqueueCustomerChargesAsync(queueId, sqsValues.PortalTypeId, sqsValues.InstanceIds, sqsValues.IsMultipleInstanceId, sqsValues.IsLastInstanceId, pageNumber, sqsValues.CurrentIntegrationAuthenticationId)

#### Database Record Updates:

MarkRecordProcessed(id, chargeId, chargeAmount, baseChargeAmount, totalChargeAmount, hasErrors, errorMessage, smsChargeId, smsChargeAmount)  
├── Update OptimizationDeviceResult\_CustomerChargeQueue:  
│ ├── SET IsProcessed = 1, ModifiedBy = 'System', ModifiedDate = GETDATE()  
│ ├── ChargeId = @chargeId, ChargeAmount = @chargeAmount  
│ ├── BaseChargeAmount = @baseChargeAmount, TotalChargeAmount = @totalChargeAmount  
│ ├── HasErrors = @hasErrors, ErrorMessage = @errorMessage  
│ ├── SmsChargeId = @smsChargeId, SmsChargeAmount = @smsChargeAmount  
│ └── WHERE Id = @id  
└── Update OptimizationMobilityDeviceResult\_CustomerChargeQueue (same structure)

## Phase 3: Completion Verification (AltaworxRevAWSCheckCustomerChargeIsProcessed)

### 3.1 Lambda Function Handler

#### Function Entry Point:

Function.FunctionHandler(SQSEvent sqsEvent, ILambdaContext context)  
├── BaseFunctionHandler(context) → KeySysLambdaContext  
├── Initialize services and repositories:  
│ ├── EnvironmentRepository()  
│ ├── OptimizationInstanceRepository(...)  
│ ├── OptimizationQueueRepository(...)  
│ ├── DeviceCustomerChargeQueueRepository(...)  
│ ├── CustomerChargeListFileService()  
│ ├── CustomerChargeListEmailService(...)  
│ ├── S3Wrapper(...)  
│ └── CheckIsProcessedService(...)  
├── Create event handler: CheckIsProcessedEventHandler(...)  
├── Create SqsValues: new SqsValues(logger, sqsEvent.Records[0])  
└── eventHandler.HandleEventAsync(sqsEvent, sqsValues)

### 3.2 Event Handler Processing

#### CheckIsProcessedEventHandler.HandleEventAsync:

HandleEventAsync(sqsEvent, sqsValues)  
├── ProcessEventAsync(sqsEvent, sqsValues)  
│ ├── Validate single record: sqsEvent.Records.Count == 1  
│ └── ProcessEventRecordAsync(sqsEvent.Records[0], sqsValues)

#### ProcessEventRecordAsync:

ProcessEventRecordAsync(message, sqsValues)  
├── Check message attributes:  
│ ├── If contains "QueueId":  
│ │ ├── Parse queueId: long.Parse(message.MessageAttributes["QueueId"].StringValue)  
│ │ ├── Get queue: \_optimizationQueueRepository.GetQueue(queueId)  
│ │ ├── Get instance: \_optimizationInstanceRepository.GetInstance(queue.InstanceId)  
│ │ └── \_checkIsProcessService.ProcessQueueAsync(queueId, instance, sqsValues)  
│ └── If contains "FileId":  
│ ├── Parse fileId: int.Parse(message.MessageAttributes["FileId"].StringValue)  
│ └── \_checkIsProcessService.ProcessQueueAsync(fileId, sqsValues)

### 3.3 Processing Verification

#### CheckIsProcessedService.ProcessQueueAsync (Queue-based):

ProcessQueueAsync(queueId, instance, sqsValues)  
├── Get environment variables and service providers  
├── Determine customer type: isNonRevCustomer = instance.AMOPCustomerId != null && instance.RevCustomerId == null && instance.IntegrationAuthenticationId == null  
├── Check if queue has more items: \_customerChargeQueueRepository.QueueHasMoreItems(queueId, isNonRevCustomer)  
├── If no more items:  
│ ├── Get charge list: \_customerChargeQueueRepository.GetChargeList(queueId)  
│ ├── Generate file: \_chargeListFileService.GenerateChargeListFile(chargeList, instance.BillingPeriodStartDate, instance.BillingPeriodEndDate, serviceProviderList)  
│ ├── Upload to S3: \_s3Wrapper.UploadAwsFile(chargeListFileBytes, fileName)  
│ ├── Wait for upload: \_s3Wrapper.WaitForFileUploadCompletion(fileName, 300 seconds, \_logger)  
│ ├── Send email based on instance type:  
│ │ ├── Single instance: \_customerChargeListEmailService.SendEmailSummaryAsync(queueId, instance, chargeListFileBytes, fileName, errorCount, isNonRevCustomer)  
│ │ └── Multiple instances (if last): ProcessSendEmailSummaryForMultipleInstanceStep(...)  
│ └── Else (items still processing):  
│ ├── Check retry count: sqsValues.RetryNumber > NUMBER\_OF\_RETRIES  
│ ├── If not exceeded: EnqueueCheckCustomerChargesIsProcessedAsync(queueId, ..., retryNumber: sqsValues.RetryNumber + 1)  
│ └── Else: Log error

### 3.4 File Generation Process

#### CustomerChargeListFileService.GenerateChargeListFile:

GenerateChargeListFile(chargeList, billingPeriodStartDate, billingPeriodEndDate, serviceProviders)  
├── Create MemoryStream and StreamWriter  
├── WriteChargeListFileHeader(sw):  
│ └── Write: "MSISDN\tIsSuccessful\tChargeId\tChargeAmount\tSMSChargeId\tSMSChargeAmount\tBillingPeriodStart\tBillingPeriodEnd\tDateCharged\tErrorMessage"  
├── WriteChargeListFileBody(sw, chargeList, billingPeriodStartDate, billingPeriodEndDate, serviceProviders):  
│ ├── For each charge in chargeList:  
│ │ ├── Get integrationId: serviceProviders.FirstOrDefault(x => x.Id == charge.ServiceProviderId).IntegrationId  
│ │ ├── Build billing period: RevIOHelper.BuildBillingPeriodDay(integrationId, billingPeriodStartDate, billingPeriodEndDate)  
│ │ └── WriteChargeRow(sw, charge, billingPeriodDay.Item1, billingPeriodDay.Item2)  
│ └── WriteChargeListFileFooter(sw, chargeList):  
│ └── Write total charges summary  
├── Flush and read bytes  
└── Return byte array

#### WriteChargeRow:

WriteChargeRow(sw, charge, billingPeriodStart, billingPeriodEnd)  
├── Calculate isSuccessful: charge.IsProcessed && (charge.ChargeId > 0 || charge.SmsChargeId > 0)  
├── Set chargeId: isSuccessful ? charge.ChargeId.ToString() : string.Empty  
├── Set smsChargeId: isSuccessful ? charge.SmsChargeId.ToString() : string.Empty  
├── Clean error message: charge.ErrorMessage.Replace('\r', ' ').Replace('\n', ' ').Replace('\t', ' ')  
└── Write line: "{charge.MSISDN}\t{isSuccessful}\t{chargeId}\t{charge.ChargeAmount}\t{smsChargeId}\t{charge.SmsChargeAmount}\t{billingPeriodStart}\t{billingPeriodEnd}\t{charge.ModifiedDate}\t{errorMessage}"

### 3.5 S3 Upload Process

#### S3Wrapper.UploadAwsFile and WaitForFileUploadCompletion:

UploadAwsFile(chargeListFileBytes, fileName)  
├── Upload file to S3 bucket: CustomerChargesS3BucketName  
├── Filename format: {queueId}.txt or {fileId}.txt  
└── Return upload result  
  
WaitForFileUploadCompletion(fileName, timeoutSeconds, logger)  
├── Poll S3 for file existence  
├── Timeout after 5 minutes (300 seconds)  
├── Return: (isUploadSuccess: bool, errorMessage: string)

### 3.6 Email Summary Process

#### CustomerChargeListEmailService.SendEmailSummaryAsync:

SendEmailSummaryAsync(queueId, instance, chargeListFileBytes, fileName, errorCount, isNonRev)  
├── Get customer name:  
│ ├── If isNonRev: \_customerRepository.GetNonRevCustomerName(instance.AMOPCustomerId)  
│ └── Else: \_customerRepository.GetCustomerName(instance.RevCustomerId)  
├── Get settings: \_settingsRepository.GetGeneralProviderSettings()  
├── Create email client: \_emailServiceFactory.getClient(credentials, RegionEndpoint.USEast1)  
├── Build email message:  
│ ├── From: generalSettings.CustomerChargeFromEmailAddress  
│ ├── To: generalSettings.CustomerChargeToEmailAddresses.Split(';')  
│ ├── Subject: generalSettings.CustomerChargeResultsEmailSubject  
│ └── Body: BuildResultsEmailBody(queueId, instance, customerName, chargeListFileBytes, fileName, errorCount)  
├── Attach file: chargeListFileBytes as tab-separated-values  
└── Send email: client.SendRawEmailAsync(sendRequest)

### 3.7 Multiple Instance Email Processing

#### ProcessSendEmailSummaryForMultipleInstanceStep:

ProcessSendEmailSummaryForMultipleInstanceStep(sqsValues, instance, proxyUrl, bucketName, queueId, isNonRev)  
├── Check if other instances still processing: \_customerChargeQueueRepository.VerifyAnyInstanceStillInProgress(instance.OptimizationSessionId.ToString(), sqsValues.PortalTypeId, isNonRev)  
├── If all completed or retry limit exceeded:  
│ └── SendMailSummaryCustomerChargeForMultipleInstance(sqsValues, instance, proxyUrl, bucketName, isNonRev)  
└── Else:  
 ├── If retry limit not exceeded:  
 │ └── EnqueueCheckCustomerChargesIsProcessedAsync(queueId, ..., customDelayTime: 900 seconds, retryNumber: sqsValues.RetryNumber + 1)  
 └── Else: Log error

#### SendMailSummaryCustomerChargeForMultipleInstance:

SendMailSummaryCustomerChargeForMultipleInstance(sqsValues, instance, proxyUrl, bucketName, isNonRev)  
├── Get queue list: \_customerChargeQueueRepository.GetQueueIsNeedSendMailSumary(sqsValues.InstanceIds, sqsValues.PortalTypeId)  
├── Get customer information:  
│ ├── If isNonRev: \_customerRepository.GetNonRevCustomers(customerIds)  
│ └── Else: \_customerRepository.GetCustomers(revCustomerGuidIds)  
├── Build email model: RevCustomerChargeEmailModel  
├── Send via proxy: client.CustomerChargeSendEmailProxy(proxyUrl, payload, \_logger)

## Database Operations Summary

### Stored Procedures Used:

1. **GET\_OPTIMIZATION\_INSTANCE**: Retrieves optimization instance details
2. **usp\_Optimization\_EnqueueCustomerCharges**: Enqueues M2M customer charges to database
3. **usp\_Optimization\_Mobility\_EnqueueCustomerCharges**: Enqueues Mobility customer charges to database
4. **GET\_OPTIMIZATION\_DEVICE\_RESULT\_CUSTOMER\_CHARGE\_QUEUE**: Gets M2M device list for processing
5. **GET\_OPTIMIZATION\_MOBILITY\_DEVICE\_RESULT\_CUSTOMER\_CHARGE\_QUEUE**: Gets Mobility device list for processing
6. **DEVICE\_CUSTOMER\_CHARGE\_QUEUE\_GET\_CHARGE\_LIST**: Gets charge list for file generation
7. **MOBILITY\_DEVICE\_CUSTOMER\_CHARGE\_QUEUE\_GET\_CHARGE\_LIST**: Gets mobility charge list for file generation

### Key Database Tables:

1. **OptimizationInstance**: Stores optimization run instances
2. **OptimizationQueue**: Stores optimization queue details
3. **OptimizationCommGroup**: Communication groups for instances
4. **OptimizationDeviceResult\_CustomerChargeQueue**: M2M device charge queue
5. **OptimizationMobilityDeviceResult\_CustomerChargeQueue**: Mobility device charge queue
6. **CustomerCharge\_UploadedFile**: File upload tracking
7. **OptimizationDeviceResult**: M2M optimization results
8. **OptimizationMobilityDeviceResult**: Mobility optimization results

## SQS Message Flow Summary

### Message Attributes Throughout the Flow:

#### Phase 1 (Controller → Enqueue Lambda):

* **InstanceId**: Instance to process
* **IsMultipleInstanceId**: 1 if multiple instances, 0 if single
* **IsLastInstanceId**: 1 if last instance, 0 otherwise
* **InstanceIds**: Comma-separated list of all instance IDs
* **CurrentIntegrationAuthenticationId**: Rev.io authentication ID

#### Phase 2 (Enqueue → Create Lambda):

* **QueueId**: Queue to process
* **IsMultipleInstanceId**: Carried forward
* **IsLastInstanceId**: Carried forward
* **InstanceIds**: Carried forward
* **PortalTypeId**: Portal type (0=M2M, 2=Mobility, CrossProvider)
* **CurrentIntegrationAuthenticationId**: Carried forward
* **PageNumber**: Page number for pagination (default 1)
* **RetryCount**: Retry attempt count (default 0)
* **RetryNumber**: Retry number for check lambda

#### Phase 3 (Create → Check Lambda):

* **QueueId** or **FileId**: Queue or file to verify
* **IsMultipleInstanceId**: Carried forward
* **IsLastInstanceId**: Carried forward
* **InstanceIds**: Carried forward
* **PortalTypeId**: Carried forward
* **CurrentIntegrationAuthenticationId**: Carried forward
* **IsSendSummaryEmailForMultipleInstanceStep**: Email flag
* **RetryNumber**: Incremented retry counter

## Error Handling and Retry Logic

### Retry Mechanisms:

1. **Rev.io API Rate Limiting (429)**:
   * Automatic retry with exponential backoff
   * Max retry count: 3 attempts
   * Re-enqueue message with RetryCount + 1
2. **Processing Failures**:
   * Mark records as failed after max retries
   * Send error notification emails
   * Continue processing other devices
3. **Check Lambda Retries**:
   * 15-minute delay between retry attempts
   * Max retry count before giving up
   * Handles incomplete processing scenarios

### Email Notifications:

1. **Success Summary**: Sent when all charges processed successfully
2. **Error Summary**: Sent when errors occur during processing
3. **Multiple Instance Summary**: Consolidated email for multiple instances
4. **Upload Error Notification**: Sent for device processing errors

## Key Integration Points

### Rev.io API Calls:

1. **GetServicesAsync**: Lookup service by MSISDN/service number
2. **AddChargeAsync**: Create customer charge in Rev.io system

### AWS Services Used:

1. **SQS**: Message queuing between lambda functions
2. **S3**: File storage for charge list results
3. **SES**: Email delivery for summaries and notifications
4. **Lambda**: Serverless function execution

### Database Integration:

1. **Connection String**: Retrieved from environment variables
2. **Stored Procedures**: Used for bulk operations and data retrieval
3. **Transactional Updates**: Ensure data consistency during processing
4. **Pagination**: 50 devices per page for efficient processing

This flow ensures reliable, scalable processing of customer charges with proper error handling, retry mechanisms, and comprehensive reporting.